Projecting Code Changes onto Execution Traces to Support Localization of Recently Introduced Bugs

Johannes Bohnet, Stefan Voigt, Jürgen Döllner
Hasso-Plattner-Institute at the University of Potsdam, Germany
{bohnet, voigt, doellner}@hpi.uni-potsdam.de

ABSTRACT

Working collaboratively on complex software systems often leads to situations where a developer enhances or extends system functionality, thereby however, introducing bugs. At best the unintentional changes are caught immediately by regression tests. Often however, the bugs are detected days or weeks later by other developers noticing strange system behavior while working on different parts of the system. Then it is a highly time-consuming task to trace back this behavior change to code changes in the past. In this paper we propose a technique for identifying the recently introduced change that is responsible for the unexpected behavior. The key idea is to combine dynamic, static, and code change information on the system to reduce the possibly great amount of code modifications to those that may affect the system while running its faulty behavior. After having applied this massive automated filtering step, developers receive support in semi-automatically identifying the root cause change by means of a trace exploration frontend. Within multiple synchronized views, developers explore when, how and why modified code locations are executed. The technique is implemented within a prototypical analysis tool that copes with large (> MLOC) C/C++ software systems. We demonstrate the approach by means of industrial case studies.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging – *Debugging aids, Tracing*.

General Terms

Verification.

Keywords

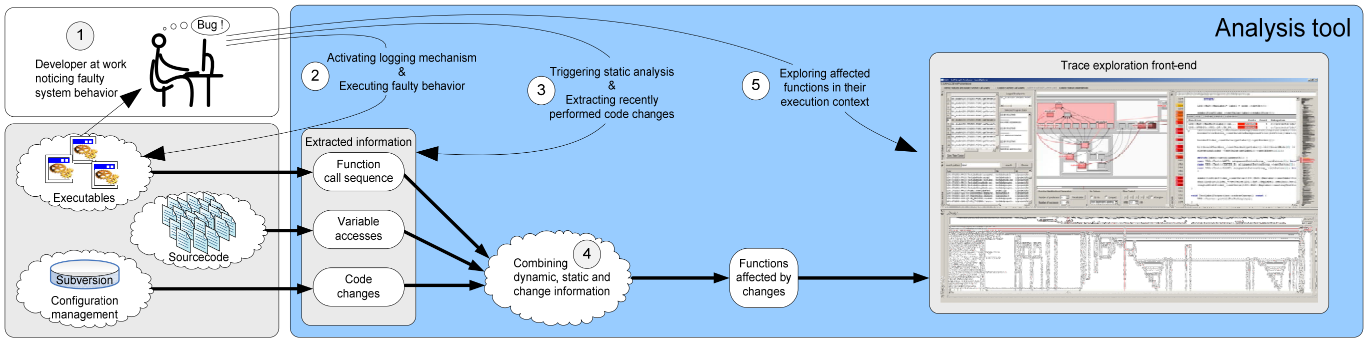
Fault Localization, Dynamic Analysis, Software Visualization.

1. INTRODUCTION

Long-living software systems tend to be complex in various ways. (1) They are typically developed collaboratively by multiple changing programmers. (2) System implementation often consists of 100.000 up to several million lines-of-code (LOC). (3) The system experiences a long time of being modified and extended. This process of software aging typically becomes manifested in design anomalies [11]. Architectural guidelines from design time describing "not-allowed-to-use"-relations between modules, for instance, are violated in some ways resulting in undocumented and hard to reveal couplings between code artifacts [8]. In consideration of the human beings' cognitive limits, it is impossible for a single developer to gather a detailed understanding of the complete system's structure and behavior. Therefore, a typical strategy to cope with the system's complexity is to focus program comprehension activities on those parts of the system implementation that the developer is currently working on [10]. Due to time pressure, the developer has to start modifying the code when his/her understanding of the system is deep enough for the given task, e.g., adding a new feature or fixing a bug. However, with this incomplete knowledge it is difficult to assure that the rest of the system's behavior is preserved. Executing a solid base of regression tests with high coverage would be a solution. Unfortunately, most systems lack of this opportunity.

Bugs introduced via side-effects pop up days or weeks later when affected system functionality is executed. Then either the testing team performing regression tests on application-level or other programmers notice the altered system behavior while doing their daily programming work. In both cases it often takes considerable work to pinpoint the source of behavior change since the executed code parts are typically scattered throughout the whole system implementation. M. C. Feathers describes the localization problem when analyzing broken application-level tests as follows: *As tests get further from what they test, it gets harder to determine what test failure means. You have to look at the test inputs, look at the failure, and determine where along the path from inputs to outputs the failure occurred* [6]. In short: Developers need to identify and follow the executed control flow path, thereby looking for places that have been modified. In practice, this is a tedious task as even during a short time of execution, thousands of functions may be active.

In this paper we propose a technique that supports developers of C/C++ software systems in localizing erroneously performed code changes that unintentionally break system behavior. The contributions of the paper are twofold:
(1) We propose a massive automated filtering technique that combines results from function call logging at runtime with data from a software configuration management system (SCM). Thereby, the possibly large amount of recently modified code is reduced to those few code locations that do affect the system while running the faulty behavior.
(2) We demonstrate the use of a trace exploration frontend that supports developers in understanding the impact of code changes that are identified during the filtering step. Multiple views permit to experience post-mortemly how functions that are affected by a code change are executed, and how by this, the change impacts on system behavior. The technique is implemented as a prototypical analysis tool that copes with large (> MLOC) C/C++ production



Analysis tool
1
Developer at work noticing faulty system behavior
Bug !
2
Activating logging mechanism & Executing faulty behavior
3
Triggering static analysis & Extracting recently performed code changes
5
Exploring affected functions in their execution context
Trace exploration front-end
Executables
Sourcecode
Subversion
Configuration management
Extracted information
Function call sequence
Variable accesses
Code changes
4
Combining dynamic, static and change information
Functions affected by changes

Figure 1: The analysis process for identifying recently performed code changes that unintentionally cause faulty system behavior.

code. We demonstrate the approach by means of industrial case studies.

2. RELATED WORK

Various automatic techniques exist to localize faults when having a set of regression tests at hand. Some approaches calculate the difference of statement coverage between passing and failing test cases [1, 13]. The identified code locations then serve as starting points for manually performing fine-grained code analysis based on the system dependency graph (SDG) [7]. Jones et al. [9] calculate a suspiciousness value for every line of code, based on their execution frequencies in passing and failing test cases. To support developers in identifying highly suspicious code lines, they additionally propose a visualization technique that encodes the suspiciousness value into color. Cleve and Zeller [5] propose a technique that runs both a passing and a failing test case in a debugger. The system is stopped at multiple code locations and part of its memory state is swapped between the two runs until the smallest memory state is isolated that causes the test case to fail. The technique is repeated at several code locations to identify the ones that are critical for the test's outcome. These locations then serve as starting point for manual code analysis using the SDG in a similar way than proposed in [13]. All these approaches differ from ours in that they operate on different input, namely a set of test cases. Our approach however exploits code change information stored in a SCM and combines it with runtime information from a single run. An approach that uses code change information to localize the root cause for a failing test case is proposed by Ren et al. [12]. Their idea is to model recently introduced changes as a set of atomic changes such as *add class* or *delete method*. Then, a semi-automated process is started where first an intermediate executable is built by applying different sets of atomic changes to the original version of the system. Second, the test case is applied onto this intermediate version of the system. By this, the minimal set of atomic changes that causes the test case to fail is isolated. The approach differs from ours in that we explicitly map runtime information onto code change data to filter irrelevant changes.

With respect to all mentioned approaches: Our approach is rather complementary to the above mentioned ones. Extending them with the ideas presented in this paper could reasonably improve them, namely by (1) mapping code change data onto executed parts of the system implementation and (2) by providing means of visual exploration of *when* and *how* introduced modifications are executed.

3. BEHAVIOR AFFECTING CODE MODIFICATIONS IN C/C++ SYSTEMS

In this section, we discuss which parts of the implementation may affect a specific system behavior if they are modified. The behavior of a C/C++ system may be changed due to various reasons:

Changes within Source Code typically are:

- Code statements within a function's body that are actually executed are modified.
- Data types of variables that are accessed from executed code statements are modified.
  In C/C++, a variable's data type is either declared within a function's body or signature, if being a local variable, parameter or return value. Or it is declared outside of "function code" if being a global variable or an attribute of a class.
- Preprocessor macros are changed.

Changes outside of Source Code typically are:

- The system environment is/behaves different:
  The system communicates with other systems that behave different or persistent data that is accessed during execution is different.
- The build configuration is modified:
  Third-party libraries are exchanged or preprocessor definitions are modified.

The approach in this paper focuses on identifying changes performed within source code only. Therefore, developers should be aware of any changes that are manifested outside of source code files. Additionally, as we use a fast and therefore lightweight static code analyzer, we do not keep track of macro code dependencies [14]. Hence, if a modification of macro code is detected, the developer has to check manually whether the change is responsible for the observed change in system behavior. The key idea proposed in this paper is to exploit the fact that we do not want to analyze whether code changes affect system behavior in general. We are interested in whether a code change is responsible for a specific faulty behavior. This behavior is reflected by an execution trace[1], i.e., a sequence of function calls.

[1] The term *execution trace* is used as synonym for *sequence of function calls*. Notice that for identifying functions that are possibly affected by a recent change, we only need to know *which* functions are executed. During trace exploration, however, we particularly make use of the *sequence* information.

With it, we know all functions' code that has to be checked for modifications. Additionally, we need to check for modifications of data types of variables that are accessed from the executed code. This information can be obtained via lightweight static code analysis. Notice that we exploit runtime information and thereby drastically simplify the static analysis process because we do not have to analyze control flow dependencies. Otherwise, it would be necessary to perform a heavyweight static dependency analysis, e.g., by calculating the system dependency graph [7].

4. ANALYSIS PROCESS

Figure 1 illustrates the analysis process a developer has to perform if he/she notices unexpected system behavior. Essentially, the developer firstly has to extract dynamic, static and code change data from the system. Then the data from the different sources are combined to identify those functions that are affected by recent change. Finally, the developer explores how affected functions are executed to assess the impact of the function's modification on system behavior. The detailed steps are:

1) During his/her daily work, a developer notices that the software system is behaving in an unexpected way. He/She knows that some days or weeks ago, the system still behaved as expected.
2) The developer enables function call logging and executes the faulty behavior.
3) The developer starts the extraction mechanisms for static and code change data.
    a. A lightweight static analysis of the code is performed.
    b. All code modification from the time when the system still behaved as expected up to now are collected.
4) Dynamic, static and code change data is combined to identify those functions that are affected by a code change.
5) Within a trace exploration frontend, the developer explores details on the resulting set of affected functions. Particularly, he/she analyzes how an affected function is executed while the faulty system behavior is exercised. Thereby, the developer understands how the change impacts on the execution of other functions. Optionally, other developers are consulted that were responsible for the changes in a suspicious function.

4.1 Fact Extraction

Function Call Logging in C/C++

Basis of the technique proposed in this paper is knowledge on the sequence of function calls that are performed while the faulty system behavior is executed. For this, we implemented a non-intrusive logging mechanism that is based on a compiler's ability to automatically insert calls to a hook function at the start of every function. Typical limitations of hook function based logging mechanisms are: (1) They are impractical for use in daily work because they require a full rebuild of the system if a developer decides to enable logging. This can be very time consuming. (2) C/C++ systems usually make use of template libraries such as the Standard Template Library (STL). Functions coming from these libraries are decorated with a call to a hook function. This unnecessarily slows down logging and significantly increases the log size. To overcome these limitations we implemented a binary patching mechanism that enables/disables a function for being logged by replacing the binary code for calling the hook function by NOP assembler instructions or vice versa. This way, we provide a fast enabling/disabling mechanism that can be used in the developers' daily work.

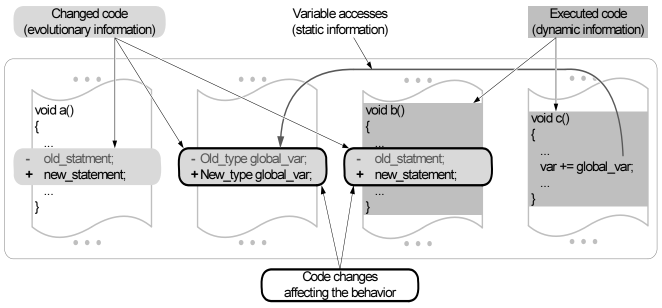


Figure 2: For detecting behavior-affecting code changes, both code of executed functions and code defining variables accessed from executed functions have to be checked for modifications.

Static Code Analysis

From function call logging (dynamic analysis) we know which functions are executed. To resolve the code lines defining data types that are accessed from the executed functions, we perform a static code analysis by exploiting the source code documentation generator tool *Doxygen* (www.doxygen.org). The Doxygen parser is able to resolve most of the C/C++ preprocessor and template programming peculiarities.

Extracting Code Change Data

Information on the changes that have been performed on the code since the time when the system last behaved as expected can easily be obtained from an SCM. Our implementation of the analysis process supports the SCM *Subversion* (subversion.tigris.org). Lines of code that are new or modified are modeled by Subversion as *added* lines in the current version. We later check for these lines and additionally check for code lines that precede deleted lines. These latter lines are necessary for detecting that formerly the control flow has entered code that is not executed anymore in the current version of the system.

4.2 Combining Data – Detecting Functions Affected by Code Changes

As discussed in Section 3, we have to check for modifications of code lines that potentially are responsible for the unexpected system behavior if they are modified (Figure 2). That is, we need to check for (1) lines implementing the functions identified by dynamic analysis, and (2) lines obtained via static analysis, i.e., lines declaring variables outside of function-related code, however, being accessed from within executed functions. If a modified code line matches a line obtained via dynamic or static analysis, the function that is dependent of the hit line is tagged as *affected by code change*. After this massive filtering step, the typically large set of executed functions is reduced to a small set that only contains the affected functions. Experience with our industrial partner (see Section 5) shows that the filtering step typically reduces the set of functions massively – primarily depending on the point in time the developer last experienced the system behaving as expected.

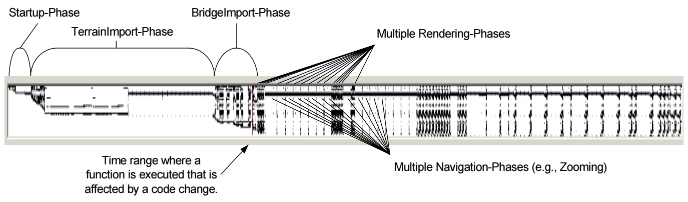


Figure 3: Depicting function activity over time permits to detect execution phases that are exercised while the system runs its faulty behavior.

4.3 Exploring Functions within their Execution Context

Within the set of functions affected by code changes, the developer needs to pinpoint the one that is responsible for the observed change in system behavior. For this, we provide a scalable and interactive trace exploration frontend. Scalable means that it copes with execution traces consisting of several tens of millions of function calls. Within multiple views the developer explores how control flow passes through the implementation while the faulty behavior is executed. Developers are able to understand the context in that a changed code location is executed. That is, they understand:

- Which execution phases are affected by a code change? During execution of the system behavior, typically different phases are passed (Figure 3). This may be phases such as a startup-phase, a phase where specific data is imported, or a phase where a specific calculation is performed.
  Knowing the phases that are affected by a code change permits to assess the change's impact and by this to prioritize which change to analyze in detail first.
- What happens along the control flow after changed code has been executed? The opportunity to analyze the function call stack at the time when modified code is executed and to discover which functions are executed afterwards permits to understand the purpose of the modified code (Figure 4). Having additionally access to the functions' source code enables developers to understand the purpose of the data that the modified code operates on.

A detailed description of the trace exploration tool is given in some of our previous publications [2, 3, 4]. Here we highlight the tool's functionality useful for interactively assessing the impact of a code change:

Multiple views (Figure 1) that focus on different trace characteristics:

Time Overview depicts different function interaction patterns and by this, visually exposes execution phases.

Detailed Time View depicts for a time range of interest which functions *are*, which *were*, and which *will be* part of the call stack. The view is similar to UML sequence diagrams; however it scales with a large amount of functions.

Call Graph View shows how functions depend on each other by calls. Functions are thereby clustered according to their containment within implementation units (files, directories). The view permits to understand collaboration patterns between functions and higher-level implementation units.

Source Code View depicts a function's code enriched with dynamic information such as resolved names of called functions.

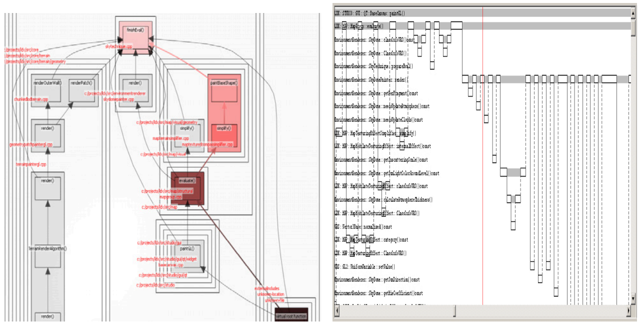


Figure 4: Multiple views on function collaboration permit to assess the impact of modifying a function.

This is of great help when polymorphism or function pointers are used in the code. Additionally, call costs and counts are shown.

As the views are **synchronized**, they permit to answer questions on time characteristics while exploring collaboration characteristics and vice versa. Developers might ask for example while analyzing the callees of a function in the call graph view: "When within the complete execution time is this callee executed? In which execution phases does it participate?"

5. CASE STUDIES

In this section we provide two case studies that we have performed together with our industrial partner 3D Geo GmbH. One of the main products 3D Geo offers is LandXplorer Studio Professional (LDX), a software solution for processing and visualizing large geodata sets (landscape and 3D city models). The LDX code is developed for more than 12 years with currently some tens of developers. LDX is written in C++ consisting of ~1.100.000 SLOC-P (7000 source files). We performed the case studies by accompanying 3D Geo's developers during their fault localization activities.

5.1 Why is the bridge getting invisible?

Figure 5 illustrates the subtle bug that a developer noticed: When zooming away from a specific building model (the bridge), some parts of it get invisible. The developer knew that approximately one month ago the bridge had been visualized correctly. Applying the analysis process described in this paper results in:

Execution trace: 1724 different executed functions.
Code Changes: 650 modified files
Combining data → affected functions: 11 functions = 0.6% of executed functions

Visually exploring affected functions within their execution context reveals that modifications concerning the bridge model import phase (Figure 3) are responsible for the unexpected behavior. The developer who performed these changes confirmed these findings.

5.2 Why is the Terrain Wizard missing?

The second case study describes the application of the approach in a situation where a developer notices that a specific wizard, the terrain loading wizard, is not shown anymore for a specific type of terrain data. One week ago, the wizard had still correctly popped up during terrain loading. We summarize the results:

Execution trace: 1674 different executed functions.
Code Changes: 65 modified files

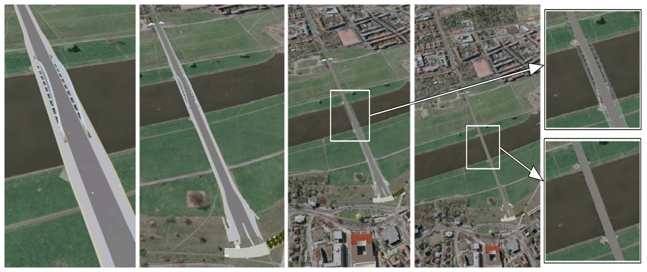


Figure 5: A developer detects a subtle bug. The bridge is getting partly invisible when zooming away from it.

Combining data → affected functions: 3 functions = 0.2% of executed functions

Analyzing when these functions are active reveals that one function is concerned with initializing the terrain data. One function is called when settings are defined how the terrain is to be visualized. The third function is called shortly after user interaction for loading the terrain took place. Code modifications within this third function turned out to be responsible for the missing terrain loading wizard.

5.3 Threats to Validity

There are two major threats to the validity of the case studies. First, we only demonstrated the approach on a single software system. Hence, to generalize the usefulness of our approach, we have to apply it on more systems from different domains. However, the case studies show that the approach copes with real problems in an industrial C/C++ system. Second, we need to increase the number of accompanied fault localization activities. To address the lack of evaluation data, we plan to incorporate our analysis tool into our industrial partner's development process. This will allow measuring in larger scale both the precision of the filtering algorithm and the effectiveness of the visualization technique.

6. CONCLUSIONS

Collaborative development of complex software systems often leads to situations where developers unintentionally break system behavior. As the change in behavior is typically noticed much later than the time of introducing the code change, it may be a tedious task to pinpoint the failure introducing change within the usually large amount of modified code. Combining data from an SCM, data obtained via static code analysis and data gathered at runtime, seems to be of great help to automatically reduce the modified code to those parts that affect the system when running the faulty behavior. Furthermore, developers seem to receive valuable support in understanding whether a modified code location may be responsible for the faulty system behavior, if they are given interactive visualizations for exploring system dynamics. Specifically, if they receive support in understanding when (execution phases) and how (function collaboration) the modified code participates during runtime.

As future work, we will enhance the visual trace exploration system by taking data dependencies into account. Firstly, the system dependency graph will be exploited such as proposed in [13, 5]. Secondly, a lightweight technique for tracing data accesses at runtime will be used. Furthermore, we will perform experiments measuring developer performance with and without the proposed technique, respectively.

Acknowledgements

We thank 3D Geo GmbH for providing us with the LandXplorer system.

7. REFERENCES

[1] H. Agrawal, J. Horgan, S. London, W. Wong. Fault Localization using Execution Slices and Dataflow Tests. *Proc. of the IEEE International Symposium on Software Reliability Engineering*, 1995, pp. 143-151.

[2] J. Bohnet, S. Voigt, J. Döllner. Locating and Understanding Features of Complex Software Systems by Synchronizing Time-, Collaboration- and Code-focused Views on Execution Traces. *Proc. of the IEEE International Conference on Program Comprehension*, 2008, pp. 266-269.

[3] J. Bohnet and J. Döllner. Analyzing Dynamic Call Graphs Enhanced with Program State Information for Feature Location and Understanding. *Proc. of the IEEE International Conference on Software Engineering*, 2008, pp. 915-916.

[4] J. Bohnet and J. Döllner. Visual exploration of function call graphs for feature location in complex software systems. *Proc. of the ACM Symposium on Software Visualization*, 2006, pp. 95-104.

[5] H. Cleve and A. Zeller. Locating causes of program failures. *Proc. of the IEEE International Conference on Software Engineering*, 2005, pp. 342-351.

[6] M. C. Feathers. Working Effectively with Legacy Code. *Prentice Hall*, 2004.

[7] S. Horwitz, T. Reps, D. Blinkley. Interprocedural Slicing Using Dependence Graphs. *SIGPLAN Notices,* vol. 23, no. 7, 1988, pp. 35-46.

[8] S. Huynh, Y. Cai, Y. Song, K. Sullivan. Automatic Modularity Conformance Checking. *Proc. of the IEEE International Conference on Software Engineering*, 2008, pp. 411-420.

[9] J. Jones, M. J. Harrold, J. Stasko. Visualization of test information to assist fault localization. *Proc. of the IEEE International Conference on Software Engineering*, 2002, pp. 467-477.

[10] A. Lakhotia. Understanding someone else's code: Analysis of experiences. *The Journal of Systems and Software*, vol. 23, no. 3, 1993, pp. 269-275.

[11] D. Parnas. Software Aging. *Proc. of the IEEE International Conference on Software Engineering*, 1994, pp. 279-287.

[12] X. Ren, O. C. Chesley, B. G. Ryder. Identifying Failure Causes in Java Programs: An Application of Change Impact Analysis. *IEEE Transaction on Software Engineering*, vol. 32, no. 9, 2006, pp. 718-732.

[13] M. Renieris and S. Reiss. Fault Localization With Nearest Neighbor Queries. *Proc. of the IEEE International Conference on Automated Software Engineering*, 2003, pp. 30-39.

[14] L. Vidacs, A. Beszedes, R. Ferenc. Macro Impact Analysis Using Macro Slicing. *Proc. of the International Conference on Software and Data Technologies*, 2007, pp. 230-235.